Lecture 5

Monday, October 7, 2024 10:01 AM

Housekeeping

Remind me to turn on zoom, share screen, start recording!

- HW 1 feedback coming soon
- HW 2 posted; due Friday 10/11 (20% penalty for Saturday night)
  - Remember to mark which problem is which on pdf
  - Remember to also hand in text file with executable SQL queries
- Project Part 1 due Friday 10/18
  - Teams should notify TAs by 10/11
- HW 3 coming soon
- Next week:
  - Mon 10/14 is Fall Break/Indigenous People's Day
  - Tues 10/15 is NYU "Legislative Monday"
  - I will post recorded lecture instead of in-class lecture
  - Office hours schedule TBD

Today:

More SQL

- Finish Chapter 3
- Chapter 4

Where are we?

Monday, February 27, 2023 10:04 AM

SELECT <attribute-list possibly including functions and/or aggregations>
FROM <table | join of tables | cartesian product of tables >
WHERE <predicate on individual rows (possibly involving …IN subquery)>
GROUP BY <attribute-list including all un-aggregated attributes from select clause>
HAVING < predicate involving results of aggregation>
ORDER BY

Tuesday, October 4, 2022 4:31 PM

NATURAL

Recap: Execution of SQL Queries

```
select dept_name, avg(salary)
from instructor join department
where budget > 100000
group by dept_name
having avg (salary) > 42000;
```

Is executed in the following steps:

- **from:** do the cross product or join (in his case) of the tables in the **from** clause.
- **where:** apply the condition in the **where** clause to every tuple of the table produced by the **from** clause.
- **group by:** aggregate and reorg table into a new table containing only the **group by** attributes and any aggregates referenced in the **having** or **select** clauses.
- **having:** apply condition in the **having** clause to every tuple of the table produced by the **group by**.
- **select:** project onto the requested attributes

Set Theory operations

Tuesday, March 10, 2020 9:38 PM

Set theory operation	**SQL Predicate**
Union ∪	UNION [can often be done with pred1 OR pred2 in the WHERE clause, instead]
Intersection ∩	INTERSECT [Not supported in MySQL]
Set Difference −	EXCEPT [MINUS in Oracle; not supported in MySQL]
Belongs to ∈	IN (subquery describing X) ∈
Does not belong to ∉	NOT IN (subquery) ∉
Is empty = ∅	NOT EXISTS (subquery)
Is not empty ≠ ∅	EXISTS (subquery)
Comparison to each element	ALL/SOME operator (subquery)

Comparison of sets to each other:

X subset Y — important operations, but not supported directly ☹

Must rewrite into complicated form

$X - Y = \emptyset \Leftrightarrow X \subseteq Y$

$X == Y$ — similar issues, even messier

Set membership

Tuesday, March 10, 2020 9:41 PM

predicate (usually in WHERE clause):

<attribute-list> IN (SELECT <attribute-list> FROM… WHERE …)
IMPORTANT: attribute lists must be same (compatible)

Saw

userID	title	Release_yr	stars
12345	Little Women	2019	5
12345	Little Women	1994	4
54321	Little Women	2019	4
54321	Finding Dory	2016	3
67890	Little Women	2019	5
67890	Finding Dory	2016	3

Find people who saw Little Women(2019) and Finding Dory (2016)

```
SELECT userID FROM Saw
WHERE title = 'LW' AND releaseYr = 2019
AND userID IN (
SELECT id FROM Saw
WHERE title = 'FD' AND relYR = 2016 )
```

predicate

Find people who saw Little Women (2019) and did not see Finding Dory (2016)

IN → NOT IN

Set difference

Aside: intersections with joins:

Tuesday, March 10, 2020 9:50 PM

Saw 1

userID	title	Release_yr	stars
12345	Little Women	2019	5
12345	Little Women	1994	4Set operations revisited
54321	Little Women	2019	4
54321	Finding Dory	2016	3
67890	Little Women	2019	5
67890	Finding Dory	2016	3

Saw 2

userID	title	Release_yr	stars
12345	Little Women	2019	5
12345	Little Women	1994	4Set operations revisited
54321	Little Women	2019	4
54321	Finding Dory	2016	3
67890	Little Women	2019	5
67890	Finding Dory	2016	3

Find people who saw Little Women(2019) and Finding Dory (2016)

```
SELECT userID
FROM   Saw AS Saw1 JOIN Saw AS Saw2
       USING (userID)
WHERE  Saw1.title = 'LW' AND
       Saw1.Release_yr = 2019
AND    Saw2.title = 'FD'
AND    Saw2.Rel_yr = 2016
```

Scalar subqueries

Tuesday, March 10, 2020 9:53 PM

- Query that is guaranteed to return a single row (no matter what valid data is in the database)
- Can be used like a value of the returned type, e.g.
  - Comparison with <, >, =, etc
  - Arithmetic operations if it's a number
- Can be used in comparisons to result of aggregation
  - Find the movie(s) that got the highest rating (from someone)

userID	title	Release_yr	stars
12345	Little Women	2019	5
12345	Little Women	1994	4
54321	Little Women	2019	4
54321	Finding Dory	2016	3
67890	Little Women	2019	5
67890	Finding Dory	2016	3

DISTINCT

```
SELECT title, ReleaseYr, MAX(stars)
FROM Saw
```

Remember that attributes in SELECT must be aggregated or in GROUP BY

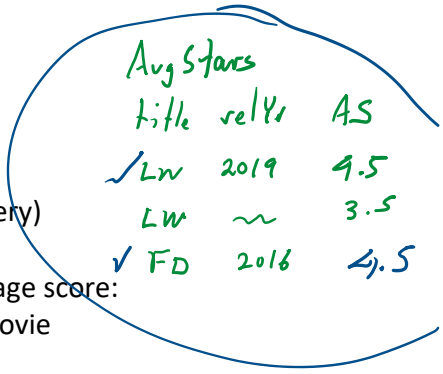
① Find max ~~~ Scalar subquery
② " movies that got that score

```
SELECT title, relYR                DISTINCT
FROM Saw
WHERE numStars =
    (SELECT max(numStars)
     FROM Saw
```

(SELECT max(numStars)
FROM Saw
)

VIEWS

Tuesday, March 10, 2020 9:58 PM



CREATE VIEW view_name AS (query)

Find movie(s) with the highest average score:

1. Find average score of each movie
2. Find max of those
3. Find title/year of the movie whose average is that max

```
CREATE VIEW avgStars AS
  ( SELECT title, relYr, AVG(numStars) AS avs
    FROM Saw
    GROUP BY title, relYR )

SELECT title, relYR
FROM avgStars
WHERE avs = (SELECT max(avs)
             FROM avgStars)
```

(1) (2) (3)

END OF Recap

Monday, February 13, 2023 4:26 PM

Correlation Variables

- Yet another way of specifying the query "Find all courses taught in both the Fall 2009 semester and in the Spring 2010 semester"

```
select course_id
from section as S
where semester = 'Fall' and year= 2009 and
        exists (select *
                from section as T
                where semester = 'Spring' and year= 2010
                        and S.course_id= T.course_id);
```

- **Correlated subquery**
- **Correlation name** or **correlation variable**

- Note: correlation can severely impact efficiency

- Subquery may use value(s) of attributes from the outer query
  - Useful for looking for pairs of tuples, t1, t2 where there is some relationship between the value of attribute in t1 and in t2
- Find people who gave a higher rating to movie X than to movie Y
- These can sometimes be done with JOIN

LW 2019 FD 2016

userID	title	Release_yr	stars (num)
12345	Little Women	2019	5
12345	Little Women	1994	4
54321	Little Women	2019	4
54321	Finding Dory	2016	3
67890	Little Women	2019	5
67890	Finding Dory	2016	3

54 321

```
SELECT userID FROM Saw AS Saw1
WHERE title = 'LW' AND Release_yr = 2019
AND userID IN
    (SELECT userID FROM Saw AS Saw2
     WHERE title = 'FD' AND Release_yr = 2016
AND
    ____________________
)
```

Alternative, with JOIN:

```
SELECT __ FROM Saw AS Saw1
    JOIN Saw AS Saw2 USING (userID)
WHERE Saw1.title ....
    AND Saw2.title ...
```

AND Saw1.numStars > Saw2.numStars

Notes on Execution of SQL Queries

- A SQL query produces a table as output
- In fact, every of the steps on previous slide starts with one or more table, and produces a table
- Thus, the meaning of nesting in the **from** clause is obvious
- For nesting in the **where**, **having**, and **select** clauses, keep in mind that these clauses are applied to each row of the table entering it
- So if there are correlation variables, the subqueries may have to be repeatedly executed
- Exception for **group by** in many DBMS: if you group by a primary key, other attributes in that table can be referred to after group-by:

select *D.dept_name*, D.budget, **avg**(salary)
from *instructor* l **join** *department D*
where *D.budget > 100000*
group by *D.dept_name*

Set Theory operations

Tuesday, March 10, 2020 9:38 PM

Set theory operation	**SQL Predicate**
Union	UNION [can often be done with pred1 OR pred2 in the WHERE clause, instead]
Intersection	INTERSECT [Not supported in MySQL]
Set Difference	EXCEPT [MINUS in Oracle; not supported in MySQL]
Belongs to	IN (subquery describing X)
Does not belong to	NOT IN (subquery)
Is empty $= \emptyset$	NOT EXISTS (subquery)
Is not empty $\neq \emptyset$	EXISTS (subquery)
Comparison to each element	ALL/SOME operator (subquery)

NOT EXISTS $= \emptyset$

EXISTS $\neq \emptyset$

Comparison of sets to each other:

X subset Y — important operations, but not supported directly ☹

Must rewrite into complicated form

$X - Y = \emptyset \Leftrightarrow X \subseteq Y$

$X == Y$ — similar issues, even messier

Test for Empty Relations

- The **exists** construct returns the value **true** if the argument subquery is nonempty.

- **exists** $r \Leftrightarrow r \neq \emptyset$
- **not exists** $r \Leftrightarrow r = \emptyset$

- NOT EXISTS (subquery) ⇔ result of subquery is empty
- This is a predicate
  - True: subquery result is empty
  - False: subquery result is not empty
- EXISTS (subquery) ⇔ result of subquery is not empty

Note: " … WHERE EXISTS (subquery) can often be done with a join instead. Think again if you're using this construct.

Monday, September 26, 2022 12:03 PM

Not Exists

- Find all students who have taken all courses offered in the Biology department.

```
select distinct S.ID, S.name
from student as S
where not exists ( (select course_id
                    from course
                    where dept_name = 'Biology' )
                   except
                    (select T.course_id
                     from takes as T
                     where S.ID = T.ID));
```

- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants

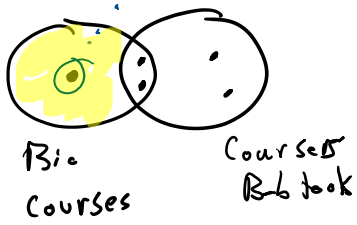
Subsets

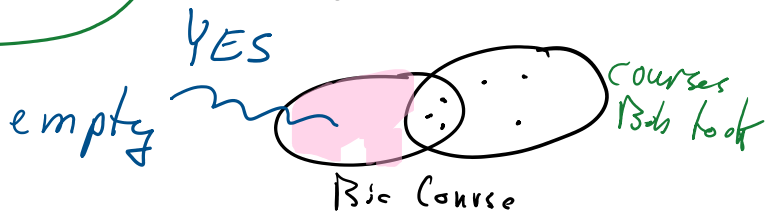
Tuesday, March 10, 2020 10:09 PM

Often want to know whether results of one subquery are all members of the result of another subquery:

Did Bob take **every** Bio course (course_id LIKE 'Bio%') ?

{bio courses} $\subseteq$ {courses Bob took}

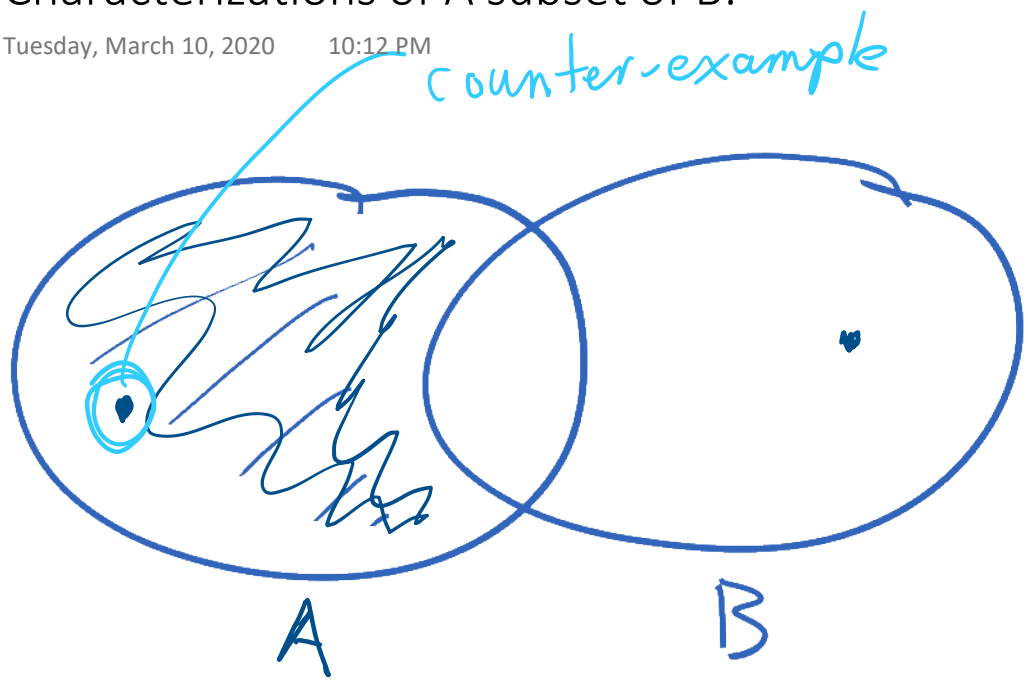




Is {bio courses} − {courses Bob took} empty ?

Characterizations of A subset of B:

Tuesday, March 10, 2020 10:12 PM



$A - B = \emptyset$ iff $A \subseteq B$

Which widgets come in every color?

Tuesday, March 10, 2020 10:13 PM

Do squares come in every color NO!

Always empty

Widgets

Colors of Square widgets

All colors

Square widgets don't come in green

How do this in SQL?

Is {all colors} - {colors of squares} = ∅

```
NOT
EXISTS
  (SELECT colors FROM widgets -- all colors
   EXCEPT
   (SELECT colors FROM widgets --
      WHERE shape = 'square' )  of square widgets
```

FALSE

What about triangles


```
NOT EXISTS
  (SELECT colors FROM widget
    EXCEPT (SELECT colors FROM
      widget WHERE
        shape = 'triangle') )
```

General question: Find shapes that come in every color

correlated query that looks at each shape in the outer query

```
SELECT DISTINCT shape FROM widget AS w
WHERE
NOT EXISTS (
  (SELECT color FROM widget)
    EXCEPT
  (SELECT color FROM widget
```

(SELECT color FROM widget
WHERE shape = w.shape))

Another approch

Tuesday, October 4, 2022 7:32 PM

$$|\{\text{bio courses}\}| = |\{\text{bio courses current student took}\}|$$

```
SELECT s.ID, c.name FROM student AS s
WHERE
   (SELECT COUNT(*) FROM course WHERE dept_name = 'Biology')

= (SELECT COUNT(DISTINCT course_id)
     FROM takes t NATURAL JOIN course
     WHERE s.ID = t.id      -- current student
        AND dept_name = 'Biology')
```

$A \subseteq B$

$x \in A \Rightarrow x \in B$

Subsets in TRC

Friday, February 23, 2024 11:17 AM

$\{$ students $s \mid x$ is a bio course $\Rightarrow s$ took $x\}$

$\{$ widgets $w \mid c$ is a color $\Rightarrow w$ comes in color $c\}$

$$\{t \mid \forall x ((x \in \text{Courses} \wedge x[\text{dept\_name}] = \text{'Bio'}) \Rightarrow \exists y \in \text{Takes} (y[\text{id}] = t[\text{id}] \wedge y[\text{course\_id}] = x[\text{course\_id}]))\}$$

Monday, February 6, 2023 10:59 AM

Test for Absence of Duplicate Tuples

- The **unique** construct tests whether a subquery has any duplicate tuples in its result.

- Find all courses that were offered at most once in 2009

```
select T.course_id
from course as T
where unique (select R.course_id
              from section as R
              where T.course_id= R.course_id
                    and R.year = 2009);
```

Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- Find the average instructors' salaries of those departments where the average salary is greater than $42,000."

```
select dept_name, avg_salary
from (select dept_name, avg (salary) as avg_salary
        from instructor
        group by dept_name)
where avg_salary > 42000;
```

- Note that we do not need to use the **having** clause
- Another way to write above query

```
select dept_name, avg_salary
from (select dept_name, avg (salary)
        from instructor
        group by dept_name) as dept_avg (dept_name, avg_salary)
where avg_salary > 42000;
```

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.

- Find all departments with the maximum budget

```
with max_budget (value) as
    (select max(budget)
     from department)
select budget
from department, max_budget
where department.budget = max_budget.value;
```

definition of max-budget dropped

- Note: often easy to output max, but hard to output who has the max

Complex Queries using With Clause

- Find all departments where the total salary is greater than the average of the total salary at all departments

```
with dept _total (dept_name, value) as
        (select dept_name, sum(salary)
         from instructor
         group by dept_name),
dept_total_avg(value) as
       (select avg(value)
       from dept_total)
select dept_name
from dept_total, dept_total_avg
where dept_total.value >= dept_total_avg.value;
```

Scalar Subquery

```
select dept_name,
        (select count(*)
           from instructor
          where department.dept_name = instructor.dept_name)
        as num_instructors
from department;
```

Modification of the Database – Deletion

- Delete all instructors

  delete from *instructor*

- Delete all instructors from the Finance department

  delete from *instructor*
  where *dept_name*= ' Finance' ;

- Delete all tuples in the *instructor* relation for those instructors associated with a department located in the Watson building.

  delete from *instructor*
  where *dept name* **in** (**select** *dept name*
  from *department*
  where *building* = ' Watson');

Example Query

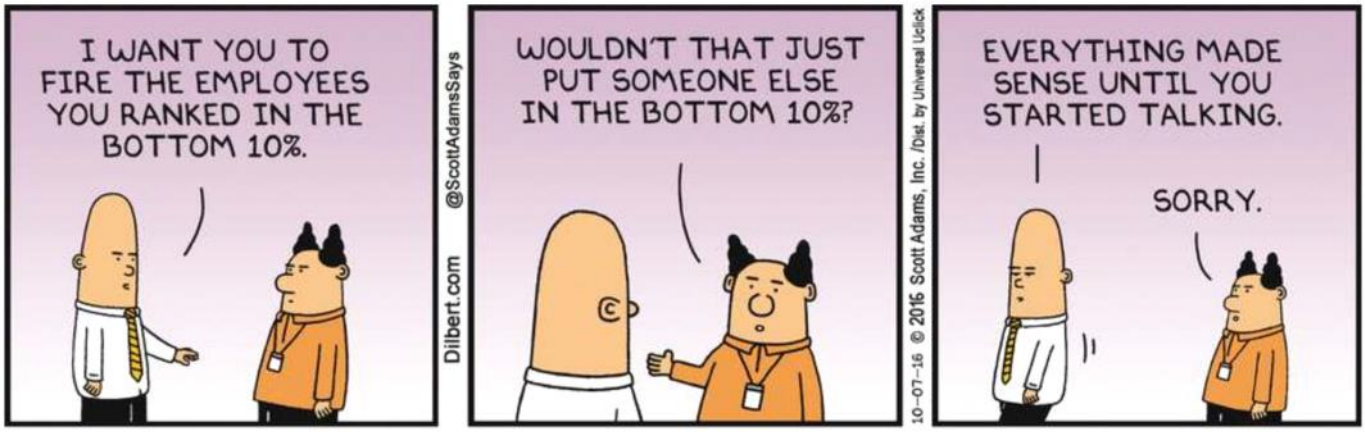
- Delete all instructors whose salary is less than the average salary of instructors

delete from *instructor*
where *salary*< (**select avg** (*salary*) **from** *instructor*);

  - Problem: as we delete tuples from deposit, the average salary changes --> what would happen?

  - Solution used in SQL:
    1. First, compute **avg** salary and find all tuples to delete
    2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Monday, February 6, 2023 10:59 AM



Monday, February 6, 2023 10:59 AM

Modification of the Database – Insertion

- Add a new tuple to *course*

```
insert into course
    values (' CS-437' , ' Database Systems' , ' Comp. Sci.' , 4);
```

- or equivalently

```
insert into course (course_id, title, dept_name, credits)
    values (' CS-437' , ' Database Systems' , ' Comp. Sci.' , 4);
```

- Add a new tuple to *student* with *tot_creds* set to null

```
insert into student
    values (' 3003' , ' Green' , ' Finance' , null);
```

Can insert multiple rows

(- - - -),
(- - - -)

Modification of the Database – Insertion

- Add all instructors to the *student* relation with tot_creds set to 0

  insert into *student*
  select *ID, name, dept_name, 0*
  from *instructor*

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like
  insert into *table*1 **select** * **from** *table*1
  would cause problems)

Monday, February 6, 2023 10:59 AM

Modification of the Database – Updates

- Increase salaries of instructors whose salary is over $100,000 by 3%, and all others receive a 5% raise
  - Write two **update** statements:

```
update instructor
   set salary = salary * 1.03
   where salary > 100000;
update instructor
    set salary = salary * 1.05
    where salary <= 100000;
```

  - The order is important (why?)
  - Can be done better using the **case** statement (next slide)

Case Statement for Conditional Updates

- Same query as before but with case statement

```
update instructor
    set salary = case
                    when salary <= 100000 then salary * 1.05
                    else salary * 1.03
                 end
```

condition

action

Updates with Scalar Subqueries

- Recompute and update tot_creds value for all students

```
update student S
  set tot_cred = ( select sum(credits)
                   from takes natural join course
                   where S.ID= takes.ID and
                         takes.grade <> 'F' and
                         takes.grade is not null);
```

- Sets *tot_creds* to null for students who have not taken any course
- Instead of **sum**(*credits*), use:

```
case
   when sum(credits) is not null then sum(credits)
   else 0
end
```

Comparison of scalar to non-scalar (Boolean aggregation operators)

Tuesday, March 10, 2020 10:00 PM

- ALL (subquery)
- SOME (subquery)
- Can do these other ways, but this is sometimes convenient
- Find instructors whose salaries are higher than *every* Comp Sci instructor

```
Select ID FROM instructor
WHERE salary >
    ALL ( SELECT salary FROM instructor
          WHERE deptname = 'Comp. Sci.' )
```

Set Comparison

- Find names of instructors with salary greater than that of some (at least one) instructor in the Biology department.

```
select distinct T.name
from instructor as T, instructor as S
where T.salary > S.salary and S.dept name =
' Biology' ;
```

- Same query using > **some** clause

```
select name
from instructor
where salary > some (select salary
from instructor
where dept name = ' Biology' );
```

- Note: could also use min here (greater than the minimum)

Definition of Some Clause

- F <comp> **some** $r \Leftrightarrow \exists\, t \in r$ such that (F <comp> t)
  Where <comp> can be: $<,\ \leq,\ >,\ =,\ \neq$

(5 < **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

(5 < **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

(= **some**) ≡ **in**
However, (≠ **some**) ≡ **not in**

Example Query

- Find the names of all instructors whose salary is greater than the salary of all instructors in the Biology department.

```
select name
from instructor
where salary > all (select salary
                    from instructor
                    where dept name = ' Biology' );
```

Definition of all Clause

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> t)

(5 < **all** {0, 5, 6}) = false

(5 < **all** {6, 10}) = true

(5 = **all** {4, 5}) = false

(5 ≠ **all** {4, 6}) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≠ **in**

Lecture 5, continued

Tuesday, October 18, 2022 3:34 PM

- Outer Joins
- Views
- Integrity Constraints
- Quick Overview of some other topics

Chapter 4: Intermediate SQL

- Join Expressions
- Views
- Transactions
- Integrity Constraints
- SQL Data Types and Schemas
- Authorization

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- A join operation is a Cartesian product which requires that tuples in the two relations match (under some condition). It also specifies the attributes that are present in the result of the join.
- The join operations are typically used as subquery expressions in the **from** clause.

Join operations – Example

- Relation *course*

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

- Relation *prereq*

course NATURAL JOIN prereq

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Note: prereq information missing for CS-315 and course information missing for CS-437.

347

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
- Uses *null* values.

Left Outer Join

- *course* **natural left outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

Note: read prere_id as prereq_id

Tuesday, October 4, 2022 4:31 PM

Right Outer Join

- *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Full Outer Join

- *course* **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Joined Relations

- **Join operations** take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- **Join condition** – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- **Join type** – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using $(A_1, A_1, \ldots, A_n)$

Joined Relations – Examples

- *course* **inner join** *prereq* **on** *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prere_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

- *course* **left outer join** *prereq* **on** *course.course_id = prereq.course_id*

course_id	*title*	*dept_name*	*credits*	*prere_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	*null*

Joined Relations – Examples

- *course* **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

- *course* **full outer join** *prereq* **using** (*course_id*)

course_id	*title*	*dept_name*	*credits*	*prere_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know an instructors name and department, but not the salary. This person should see a relation described, in SQL, by

  select *ID*, *name*, *dept_name*
  from *instructor*

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a "virtual relation" is called a **view**.

View Definition

- A view is defined using the **create view** statement which has the form

  create view *v* **as** < query expression >

  where <query expression> is any legal SQL expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

Example Views

- A view of instructors without their salary

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

- Find all instructors in the Biology department

```
select name
from faculty
where dept_name = 'Biology'
```

- Create a view of department salary totals

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

Views Defined Using Other Views

- **create view** *physics_fall_2009* **as**
  select *course.course_id*, *sec_id*, *building*, *room_number*
  from *course*, *section*
  where *course.course_id* = *section.course_id*
  and *course.dept_name* = ’ Physics’
  and *section.semester* = ’ Fall’
  and *section.year* = ’ 2009’ ;

- **create view** *physics_fall_2009_watson* **as**
  select *course_id*, *room_number*
  from *physics_fall_2009*
  where *building*= ’ Watson’ ;

Views Defined Using Other Views

- One view may be used in the expression defining another view,
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be *recursive* if it depends on itself.

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

  repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1
- As long as the view definitions are not recursive, this loop will terminate.

How Can your DBMS Maintain Views?

- If a view is often used in queries, it might be very inefficient to recompute the view from scratch every time it is used.

How Can your DBMS Maintain Views?

- If a view is often used in queries, it might be very inefficient to recompute the view from scratch every time it is used.
- To deal with this, your DBMS will often materialize and then maintain the view under updates to the underlying tables ("materialized views")
- Example 1: the faculty view on slide 12:
  - Every insert, delete, update to instructor can be translated into a suitable insert, delete, or update on the materialized view

How Can your DBMS Maintain Views?

- If a view is often used in queries, it might be very inefficient to recompute the view from scratch every time it is used.
- To deal with this, your DBMS will often materialize and then maintain the view under updates to the underlying tables ("materialized views")
- Example 1: the faculty view on slide 12:
  - Every insert, delete, update to instructor can be translated into a suitable insert, delete, or update on the materialized view
- Example 2: the departments_total_salary view on slide 12:
  - Every insert, delete, and update to instructor can be applied to the view by increasing, decreasing, or updating department salary totals
- The best strategy depends on number of view accesses versus number of updates to the underlying table

Update of a View

- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty* **values** (' 30765' , ' Green' , ' Music');

- This insertion could be represented by the insertion of the tuple

  (' 30765' , ' Green' , ' Music' , null)

  into the *instructor* relation

- Is this legal?

Update of a View

- Add a new tuple to *faculty* view which we defined earlier

  insert into *faculty* **values** (' 30765' , ' Green' , ' Music');

- This insertion could be represented by the insertion of the tuple

  (' 30765' , ' Green' , ' Music' , null)

  into the *instructor* relation

- Is this legal?
- Yes, assuming we may assign a null value to salary
- If salary attribute was NOT NULL, cannot do this

Some Updates cannot be Translated Uniquely

- **create view** *instructor_info* **as**
  select *ID*, *name*, *building*
  from *instructor*, *department*
  where *instructor.dept_name= department.dept_name*;
- **insert into** *instructor info* **values** (' 69987' , ' White' , ' Taylor');
  - which department, if multiple departments in Taylor?
  - what if no department is in Taylor?
- Most SQL implementations allow updates only on simple views
  - The **from** clause has only one database relation.
  - The **select** clause contains only attribute names of the relation, and does not have any expressions, aggregates, or **distinct** specification.
  - Any attribute not listed in the **select** clause can be set to null
  - The query does not have a **group** by or **having** clause.
- But e.g., Oracle allows some updates on views with several relations

And Some Not at All

- **create view** *history_instructors* **as**
  select *
  from *instructor*
  where *dept_name*= ’ History’ ;
- Insert (’ 25566’ , ’ Brown’ , ’ Biology’ , 100000) into *history_instructors*

Transactions

- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions
- Transactions begin implicitly
  - Ended by **commit work** or **rollback work**
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (e.g. using API)
  - In SQL:1999, can use: **begin atomic** …. **end**

Integrity Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
  - A checking account must have a balance greater than $10,000.00.
  - A salary of a bank employee must be at least $4.00 an hour.
  - A customer must have a (non-null) phone number.

Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate

single attribute
tuple of attribute

single row

check (salary > 0 and salary < 500000)

Not Null and Unique Constraints

- **not null**
  - Declare *name* and *budget* to be **not null**

    name **varchar**(20) **not null**
    budget **numeric**(12,2) **not null**

- **unique** (A_1, A_2, …, A_m)
  - The unique specification states that the attributes *A*1, *A*2, … *A*m form a candidate key.
  - Candidate keys are permitted to be null (in contrast to primary keys).

The check clause

- **check** (P)

  where P is a predicate

Example: ensure that semester is one of fall, winter, spring or summer:

```
create table section (
    course_id varchar (8),
    sec_id varchar (8),
    semester varchar (6),
    year numeric (4,0),
    building varchar (15),
    room_number varchar (7),
    time slot id varchar (4),
    primary key (course_id, sec_id, semester, year),
    check (semester in (' Fall' , ' Winter' , ' Spring' ,
        'Summer' ))
```

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
  - Example: If "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology".
- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

Tuesday, October 4, 2022 4:31 PM

Cascading Actions in Referential Integrity

- **create table** *course* (
  course_id **char**(5) **primary key**,
  title **varchar**(20),
  dept_name **varchar**(20) **references** *department*
)
- **create table** *course* (
  …
  dept_name **varchar**(20),
  foreign key (*dept_name*) **references** *department* (dept_name)
  on delete cascade
  on update cascade,
  …
)
- alternative actions to cascade: **set null**, **set default**

What happens when a violation is caused by changing the referenced table (department)

Cascade : deletia the violating tuples from referencing table (course)

Set NULL : change attribute value to NULL

Set DEFAULT

Complex Check Clauses

- **check** (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))
  - why not use a foreign key here?
- Every section has at least one instructor teaching the section.
  - how to write this?
- Unfortunately: subquery in check clause not supported by pretty much any database
  - Alternative: triggers (later)
- **create assertion** <assertion-name> **check** <predicate>;
  - Also not supported by anyone

Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

Other Features

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))
- **create index** *studentID index* **on** *student*(*ID*)
- Large objects
  - *book review* **clob**(10KB)
  - *image* **blob**(10MB), *movie* **blob**(2GB)
- **create type** construct in SQL creates user-defined type

  create type *Dollars* **as numeric (12,2) final**

  - **create table** *department*
    (*dept_name* **varchar** (20),
    building **varchar** (15),
    budget Dollars);

- **create domain** construct in SQL-92 creates user-defined domain types

  create domain *person_name* **char**(20) **not null**

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.
- **create domain** *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** (' Bachelors' , ' Masters' , ' Doctorate'));
- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
  - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
  - **clob**: character large object -- object is a large collection of char data
  - When a query returns a large object, a pointer is returned rather than the large object itself.